Regression Test Selection when Evolving Software with Aspects

Romain Delamare, Benoit Baudry, Yves Le Traon

Regression Test Selection when Evolving Software with Aspects

Romain Delamare
IRISA / INRIA Rennes
Avenue du Général Leclerc
35042 Rennes Cedex
France
romain.delamare@irisa.fr

Benoit Baudry
IRISA / INRIA Rennes
Avenue du Général Leclerc
35042 Rennes Cedex
France
benoit.baudry@irisa.fr

Yves Le Traon
ENSTB
2, rue de la Châtaigneraie
35576 Cesson Sévigné Cedex
France
yves.letraon@enst-bretagne.fr

ABSTRACT

Aspect-oriented software evolution introduces new challenges for regression test selection. When a program, that has been thoroughly tested, evolves by addition of an aspect, it is important for regression test selection to know which test cases are impacted by the new aspects and which are not. The work presented here proposes a classification for regression test cases and introduces an algorithm for impact analysis of aspects on a set of test cases. A major benefit of this analysis is that it saves the execution of test cases that are not impacted.

1. INTRODUCTION

When software evolves, it is necessary to perform regression testing in order to check that no unexpected change has been introduced. A major challenge for regression test selection techniques is to identify the subset of test cases for one version of the system that must be re executed to validate an evolution of the system. It is very important to precisely identify this subset since executing the whole test suite is very time consuming in the case of large software systems. Also, a selection technique must distinguish between the test cases that can be kept unchanged (that can be re executed as they are for regression testing) and the ones that must evolve (to take changes into account). Several techniques have been proposed for regression test selection in the context of object-oriented programs [2, 3, 5, 7]. Here we are interested in regression testing in the context of aspect-oriented software evolution.

In this paper, we propose an impact analysis of aspects on test cases, as well as a classification of regression test cases. The analysis identifies the test cases that can cover a part of the system where an aspect is woven. These test cases are called *impacted test cases*. The test cases that are not concerned by the aspect weaving are called *non-impacted test cases*. This impact analysis provides the necessary information for regression test selection.

The impact analysis is performed statically, which avoids executing all the non-impacted test cases. In the first step of the analysis, the aspect and the base program are analyzed in order to identify which base methods are impacted by the aspect's weaving. Second, the static call graph is built for each test case. The graph is then analyzed to check whether it contains an impacted method.

Once the impacted test cases have been identified, we run them in order to get a verdict for each of them. The test cases that are impacted by the change but that still pass are of particular interest for us. This verdict can be interpreted in several ways. Either the aspect has no effect on the behaviour that is evaluated by the test case; or the aspect adds behaviour without modifying the previous one. When this happens, the test case still passes but its oracle function does not evaluate the behaviour added by the aspect. The test case should then be modified and re-executed in order to validate the evolution introduced by the aspect. The test cases that are impacted and that still pass can be automatically identified, but the analysis to know if they must change has to be performed manually.

The paper is organized as follows. Section 2 describes the general principles for the impact analysis we want to perform for regression testing selection. Section 3 presents the classification of test cases and discusses the meaning of each class of test case for regression testing. Section 4 goes into the detail of our solution for regression test selection. Section 5 discusses related works and section 6 concludes.

1.1 Relevance to LATE topics

The approach presented in this paper is related to aspect evolution and software evolution with aspects. Our technique leads to better evolvable aspects because it eases the validation of the evolutions thanks to a safe regression test selection. Regression testing has two purposes in this situation: it is a necessary step for validating the evolutions; it helps getting confidence that the introduced evolutions do not disturb existing behaviour. Thus, this approach can also be used to validate the preservation of certain behaviours after the addition of a new aspect or the evolution of an aspect already woven: it selects the test cases that must be executed to validate this behaviour preservation.

2. IMPACT ANALYSIS

In this section, we define the test cases that we manipulate

```
public class Stack {
  class Cell {
    int data; Cell next;
    Cell(Cell n, int i) { next = n; data = i
        ; }
  }
  Cell head = null;
  public void push(int i) {
    head = new Cell(head,i);
  }
  public void pushArray(int[] t) {
    for(int i:t) push(i);
  }
  public int pop() {
    if(head==null)
      throw new EmptyStackError();
    int result = head.data;
    head = head.next;
    return result;
  }
}
```

Figure 1: Java Integer Stack implementation

and we introduce the global objectives of an impact analysis in case of aspect oriented evolution.

2.1 Test cases

The starting point for the impact analysis proposed in this paper is a base program with its associated unit and system test cases. Our goal is then to analyze which test cases are impacted by the introduction of an aspect in the base program. Before we define what we mean by an impacted test case, we briefly present the structure of the test cases that we consider. It is important to understand this structure in order to understand the different ways an aspect can impact a test case. The analysis we propose is dedicated to Java programs, thus the base program is implemented in Java and the test cases are implemented with JUnit. All test cases conform to the same structure defined in the following.

Test cases.

A test case is composed of three parts.

- The first part of the test case is the *preamble*, which puts the system in a certain state required by the test scenario.
- The second part actually executes the method under test with the *test data*.
- The third part is the *oracle*, composed of a set of assertions. The test case fails if any of the assertion of the oracle fails.

The following test case example is a unit test for the `pop` method of the class `Stack`:

```
1 @Test
  public void testPop2() {
3   Stack s = new Stack();
    s.add(3);
5   int i = s.pop();
    assertEquals(3,i);
7 }
```

The preamble is on lines 3 and 4: a new stack is created and an integer (3) is pushed. Line 5 exercise the method under test: `pop` is called on `s` and its result is stored in the local variable `i`. Finally line 6 is the oracle: an assertion checks if the result of `pop` equals the pushed integer.

It is important to note that in most cases the oracle (the set of assertions) cannot be exhaustive and is only partial: it is too expensive to check that the expected behavior occurred and *nothing else* occurred. For instance, in the previous example the oracle just checks if the result of `pop` is correct but it does not check if the stack is empty as it should be after the call of `pop`.

```
public class StackTest {
  Stack stack;
  @Before
  public void setUp() throws Exception {
    stack = new Stack();
  }
  @Test
  public void testPush() {
    stack.push(-3);
    assertEquals(-3,stack.pop());
  }
  @Test
  public void testPushArray() {
  int[] t = {1,2,3};
    stack.pushArray(t);
    assertEquals(3,stack.pop());
    assertEquals(2,stack.pop());
    assertEquals(1,stack.pop());
  }
  @Test
  public void testPop() {
    try {
      stack.pop();
      fail();
    } catch(EmptyStackError e)
    {assertTrue(true);}
  }
}
```

Figure 2: JUnit tests for the `Stack` class

```
public aspect NonNegativeArg {
  before(int i): execution(* *.*(int)) &&
      args(i) {
    if(i<0)  throw new Error("Negative arg")
        ;
  }
}
```

Figure 3: The `NonNegativeArg` aspects implementation

2.2 Objective of the impact analysis

The main objective of this work is to determine what test cases should be kept unchanged, what test cases should be modified, and what test cases should be removed in case of aspect oriented evolution. This analysis starts by detecting the methods in the base code that are impacted by an aspect. Then, a test case is detected as impacted if it can reach an impacted method.

Impacted method.

An impacted method is a method where an aspect is woven. Thus, a method is impacted if it includes a joinpoint shadow where an advice is woven. For instance if a pointcut matches every call to a method m, then the methods that

Impact Analysis	Execution Result	Effect
NI	No need to be executed	
I	IP	– evolve to pass – evolve to fail – no change
	IF	– evolve to pass – remove

Table 1: Summary of the test case classification

call m are impacted. If a pointcut matches every execution of m, then m is impacted.

Figure 1 shows an example that implements an integer Stack in Java (adapted from Xie *et al.* [8]) and Figure 3 displays an aspect called `NonNegativeArg`. In this example, the `NonNegativeArg` aspect impacts the `push` method because its pointcut expression matches every execution of `push`, so its advice is woven within `push`.

Impacted test case.

An impacted test case is a test case that can possibly execute at least one impacted method according to the previous definition. Let $M(tc)$ be the set of all the methods statically reachable by a test case tc and $I(a)$ the set of all the methods impacted by an aspect a. A test case tc is impacted if and only if $\exists a, M(tc) \cap I(a) \neq \emptyset$.

Figure 2 shows three JUnit test cases for the Stack class. The `push` method is impacted by the `NonNegativeArg` aspect, so the `testPush` and `testPushArray` test cases are impacted:

- `testPush` calls an impacted method
- `testPushArray` calls the `pushArray` method which calls `push` which is impacted

A test case can be impacted by an aspect in several different ways. The preamble may not achieve to put the system in the correct state. For instance in the example of section 2.1, if an aspect only allows pushing an integer in a stack under certain conditions, the integer might not be pushed as expected. The aspect might modify the behavior of the method under test and the result is not the same, so the oracle is not correct, or a behavior has been added, and the oracle must be extended. The input domain of the method under test may also change, so the input data may not be correct anymore.

Next section introduces the different classes of test cases that our analysis can identify and section 4 details the static analysis that we perform in order to determine the set of impacted test cases.

3. CLASSIFICATION OF TEST CASES

Depending on whether they are impacted or not and whether they pass or not we classify the test cases as presented in the following.

3.1 Classification definition

We define a first level for the classification:

Not Impacted(NI) and Impacted(I) test cases.

The **NI** class is the class of test cases that are not impacted by the aspect. On the contrary, the **I** class is the class of the test cases that are impacted according to the previous definition. Those two classes are exclusive and can be computed statically. At a second level of classification, the **I** class is divided into two subclasses:

Impacted Passing(IP) and Impacted Failing(IF) test cases.

The **IP** class contains the test cases that are impacted and that still pass, i.e. the assertions for the oracles are all satisfied. The **IF** contains the test cases that are impacted and fails.

This classification has two benefits. First, test cases classified as **NI** do not need to be executed for regression testing after the aspect weaving. As we discuss it in section 4, the classification of **NI** and **I** test cases is performed statically, thus the test cases that are not impacted by the aspect are never executed. Second, the distinction between **IP** and **IF** also offers valuable information for regression testing. In particular, a test case that is impacted and that still passes (**IP**) must be carefully studied. It is important to check if the test case still passes because the aspect has not changed the behavior it evaluates or if it passes because its oracle is too weak and does not consider the new behavior inserted by the aspect.

3.2 Classification exploitation

Once the test cases have been classified we can determine whether they should be modified, removed or kept unchanged. This section describes how to consider the test cases according to their class.

It is not necessary to execute the test cases of class **NI**. As we are sure that they are not impacted, if they passed before the weaving of the aspects, they should still pass. It is a major contribution as it may save a lot of time in large systems.

The test cases in class **IP** can evolve in three different ways, depending on the oracle. It is important to check the oracle because it might have become too weak. As discussed before the oracle is usually partial and some properties added by the aspect might not be checked. When it is the case, the test case can evolve to fail – assertions are added to the oracle and at least one of them fails – or it can evolve to pass – assertions are added and they all pass. When evolving, a test case checks new properties added by an aspect. If the test case fails there is probably an error in the aspect. The oracle may also not need any change and the test case is kept unchanged.

For the test cases of class **IF** the tester must focus on the input data. If the input data of a **IF** test case is still in the input domain of the method under test then the expected result must have change and the test case can evolve by correcting the oracle. If the input data is no more in the input domain, then it should be removed.

Table 1 shows a summary of the test case classification with the different possible evolutions.

In the `Stack` example, Figure 2 shows three test cases. Test case `testPush` is impacted as it calls an impacted method, `push`. When executed this test case fails as it calls `push` with a negative argument so it is in class **IF**. This test case should be removed as its input data is no more in the input domain of the method under test. Test case `testPushArray` is impacted too but still passes so it is in class **IP**. There is nothing new to check so the oracle is still good and this test case should be kept unchanged. Finally test case `testPop` is not impacted so it is kept unchanged

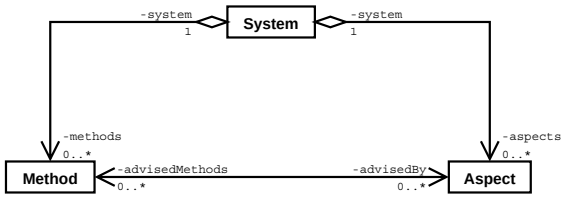

Figure 4: Metamodel of the impacted methods model.

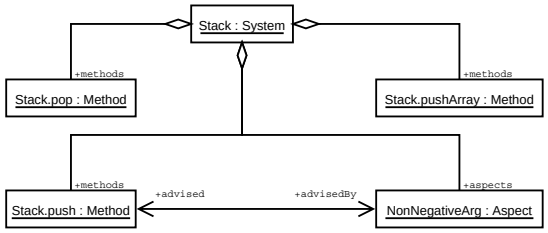

Figure 5: Object diagram of the impacted methods model of the stack example.

and not executed.

4. STATIC REGRESSION TEST SELECTION

To distinguish between **NI** and **I** test cases, we statically analyze the Java base program, the aspects and the test cases.

Thanks to the pointcut expression we can statically know in which methods the aspect is woven. The analysis thus consists of two steps. First, we build a model that captures the relationships between aspects and the base code. Then we analyze the static call graph of each test case in order to determine if they are impacted.

4.1 Impacted methods model

When a Java program and an aspect are compiled with the AspectJ compiler, the joinpoints where the aspect has to be woven are statically identified. To evaluate the impact of the aspects on the test cases we build a model that represents the relationships between the aspects and the base code.

Figure 4 shows the metamodel we have designed to capture impacted methods. A system is composed of a set of methods and a set of aspects, and there is a relation between the aspects and the methods that represents the fact that an aspect is woven *within* the code of a method.

We have extended the AspectJ compiler in order to build an instance of this metamodel. The AspectJ compiler compiles the Java classes and weaves the aspects, directly generating woven byte code. The compiler offers an interface to add a build listener (which is called after each compilation). The compiler then provides a list of weaving relations between the aspects and the different parts of the Java code where the aspects are woven. We use those information to build the impacted methods model.

As an example, figure 5 shows the model corresponding to the stack example, in the form of an object diagram. This model contains an instance of class `Method` for each method of `Stack` and an instance of `Aspect` for the aspect `NonNegativeArg`. As seen before, the advice of `NonNegativeArg` is woven within `Stack.push`, so there

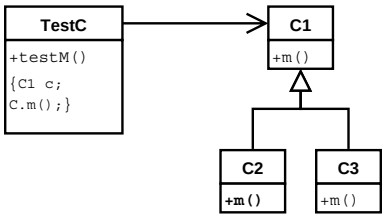

Figure 6: Class diagram illustrating the overapproximation with polymorphism.

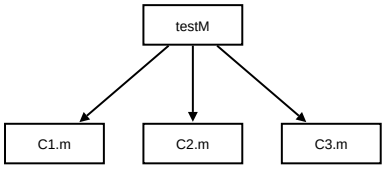

Figure 7: The static call graph of `testM`.

is a relationship between the two instances.

4.2 Static call graph

The second step consists in checking if a test case can reach an impacted method or not. This requires knowing the methods that are reachable by the test case. The adopted solution relies on a static call graph that allows us to know which methods are reachable by a test case.

Static call graph.

A static call graph is a pair (N, A) where:

- N is a set of nodes that model methods in the system, each method being represented by only one node.
- $A \subset N \times N$ is a set of edges that represent potential calls. If $n1, n2 \in N$ respectively represents the methods `m1` and `m2`, then $n1 \times n2 \in A$ means that `m1` potentially calls `m2`..

The notion of potential call is related to polymorphism. Figure 6 shows a class diagram where `C1` is a class defining a method `m` and `C2` and `C3` are two classes inheriting from `C1` and overriding `m`. The testing class `TestC` has a test that calls `m` on an object declared with class `C1`, but in the general case we cannot know statically know which is the actual type of the object at runtime and which method is actually executed. That is why `testM` can potentially call `C1.m`, `C2.m` and `C3.m`. Figure 7 shows the static call graph of `testM`.

To determine if a test case is impacted we then traverse its static call graph. For each encountered method we check refer to the impacted methods model to know if it is impacted by an aspect. For instance in the static call graph of Figure 8, which shows the static call graph of the `testPushArray` test case, the `Stack.push` method is called, and the impacted methods model from Figure 5 shows that there is an aspect woven within `Stack.push`, so the `testPushArray` is impacted.

In the example of Figure 6, if we consider that only `C2.m` is impacted, as it is reachable by `testM` (Figure 7), we consider `testM` as impacted. This is an overapproximation as we cannot be sure that `C2.m` is actually executed by `testM`.

This overapproximation of the methods reachable by a test case allows us to guarantee that the test cases detected

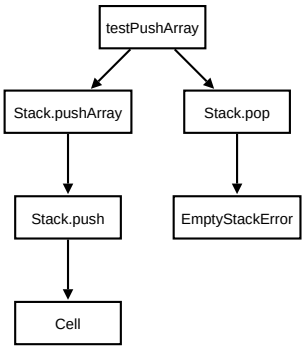

Figure 8: Example of static call graph, corresponding to the `testPushArray` test case.

as non-impacted are actually not impacted. This overapproximation is also the only way to perform a static impact analysis that allows us to never execute the non-impacted test cases. This overapproximation also ensures that all impacted test cases will be detected. Thus, this static analysis allows us to perform a safe regression testing selection as defined by Rothermel *et al.* [6]. All test cases that have to be executed for regression testing are identified in the set of **I** test cases.

5. RELATED WORKS

AOSD introduces a number of challenges for testing. For instance it is difficult to unit test an advice as it should be woven within a base program to be executed. For this reason the testing techniques for object-oriented programs cannot be directly applied on AOSD programs.

Xu *et al.* [10] have extended a regression test selection technique for Java introduced by Harrold *et al.* [1] to take aspects into account. Their approach is based on a specialized control-flow graph, called AJIG that extends the JIG introduced by Harrold *et al.*. This approach requires the construction and traversal of two CFGs, one for each version of the program.The impacted edge are identified during the traversal, and if a test case exercise an impacted edge it is rerun. The contribution of this work is the identification of the test cases that do not need to be executed, which our approach allows too, but on the contrary to our work they do not consider the evolution of the impacted test cases.

Xie *et al.* [9] have introduced a framework for detecting redundant unit tests in AspectJ programs. Aspects are usually unit tested in the context of the impacted classes as it not possible to execute them alone. So it is possible to use automated test generators for Java to test AspectJ aspects, but they produced a lot of redundant test cases. The framework proposed by Xie *et al.* removes the test cases that do not exercise a new behavior. This approach could be applied to regression testing as the introduction of an aspect can introduce redundancy in the test cases.

6. CONCLUSION

In this paper we have presented a classification for regression test cases in case of aspect-oriented evolution. This classification distinguishes the test cases that are not impacted by the aspects (class **NI**) from the test cases that are impacted by the aspects (class **I**). We have also proposed a static analysis to automate the impact analysis of aspect weaving on a set of test cases. This analysis takes advantage of the pointcut expression to evaluate the impact of aspects on the test cases.

In future work, we will completely implement the impact analysis in order to experiment with our test regression selection technique on aspect-oriented programs. We also plan to investigate a refinement of the test cases classification using the aspect classification of Rinard *et al.* [4]. This would improve the evolution of the test cases and the fault localization. For instance the test cases impacted by an augmentation aspect should be executed but they must pass, otherwise we are sure that the fault is localized in the aspect. Also, a a better understanding of the impacts of an aspect will probably help the evolution of the test cases.

7. REFERENCES

[1] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *OOPSLA'01: Proceedings of the 16th conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 312–326, 2001.

[2] H. K. N. Leung and L. White. Insights into regression testing. In *ICSM'89: Proceedings of the 5th International Conference on Software Maintenance*, pages 60–69, 1989.

[3] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. *SIGSOFT Software Engineering Notes*, 29(6):241–251, 2004.

[4] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *FSE'04: Proceedings of the 12th international symposium on Foundations of Software Engineering*, pages 147–158, 2004.

[5] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEE Transactions on Software Engineering*, 22(8):529–551, August 1996.

[6] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering Methodology*, 6(2):173–210, 1997.

[7] F. I. Vokolos and P. G. Frankl. Pythia: a regression test selection tool based on textual differencing. In *3rd International Conference on Reliability, Quality and Safety of Software-Intensive Systems*, pages 3–21, London, UK, UK, 1997. Chapman & Hall, Ltd.

[8] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of AspectJ programs. In *AOSD'06: Proceedings of the 5th international conference on Aspect-Oriented Software Development*, pages 190–201, 2006.

[9] T. Xie, J. Zhao, D. Marinov, and D. Notkin. Detecting redundant unit tests for AspectJ programs. In *ISSRE'06: Proceedings of the 17th International Symposium on Software Reliability and Engineering*, pages 179–190, 2006.

[10] G. Xu and A. Rountev. Regression test selection for AspectJ software. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 65–74, 2007.